PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 4 : G06F 5/00 — A1

(11) International Publication Number: WO 90/01738

(43) International Publication Date: 22 February 1990 (22.02.90)

(21) International Application Number: PCT/US89/02994

(22) International Filing Date: 10 July 1989 (10.07.89)

(30) Priority data:
226,078 29 July 1988 (29.07.88) US

(71) Applicant: HUNTER SYSTEMS SOFTWARE, INC. [US/US]; 444 Castro Street, Mountain View, CA 94041 (US).

(72) Inventors: HUNTER, Colin, B. ; 1200 Hamilton Avenue, Palo Alto, CA 94301 (US). BANNING, John, P. ; 808 Ticonderoga, Sunnyvale, CA 94087 (US). PUFAL, Hans ; 444 Castro Street, Mountain View, CA 94041 (US).

(74) Agents: YIN, Ronald, L. et al.; Limbach, Limbach & Sutton, 2001 Ferry Building, San Francisco, CA 94111 (US).

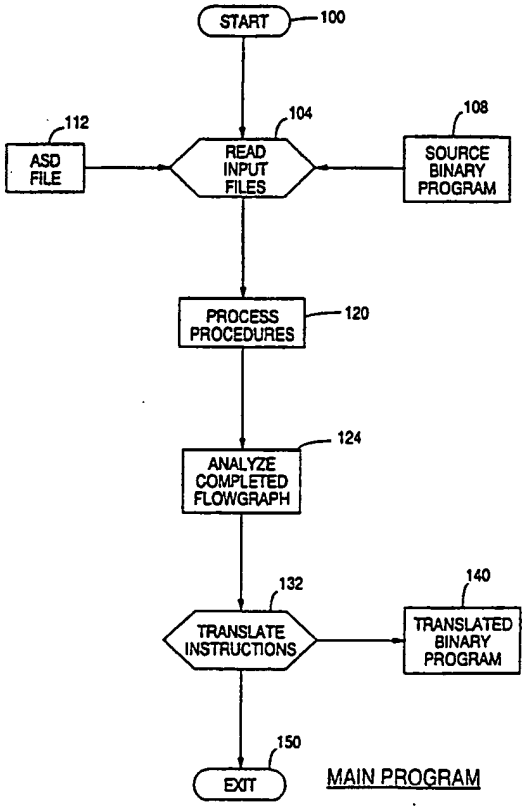
(81) Designated States: AT (European patent), BE (European patent), CH (European patent), DE (European patent), FR (European patent), GB (European patent), IT (European patent), JP, LU (European patent), NL (European patent), SE (European patent).

Published
With international search report.

(54) Title: MACHINE PROCESS FOR TRANSLATING PROGRAMS IN BINARY MACHINE LANGUAGE INTO ANOTHER BINARY MACHINE LANGUAGE

(57) Abstract

A machine process is disclosed in which a first program in one binary language is translated into a second program in another binary machine language. The process disassembles the first binary program (120), analyzes the first program to produce global flow analysis data (124), uses this global flow analysis data to complete the disassembly, and produces a translated binary machine language version of the first program, using the global flow analysis data to generate the second program (140).



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	ES	Spain	MG	Madagascar
AU	Australia	FI	Finland	ML	Mali
BB	Barbados	FR	France	MR	Mauritania
BE	Belgium	GA	Gabon	MW	Malawi
BF	Burkina Fasso	GB	United Kingdom	NL	Netherlands
BG	Bulgaria	HU	Hungary	NO	Norway
BJ	Benin	IT	Italy	RO	Romania
BR	Brazil	JP	Japan	SD	Sudan
CA	Canada	KP	Democratic People's Republic of Korea	SE	Sweden
CF	Central African Republic			SN	Senegal
CG	Congo	KR	Republic of Korea	SU	Soviet Union
CH	Switzerland	LI	Liechtenstein	TD	Chad
CM	Cameroon	LK	Sri Lanka	TG	Togo
DE	Germany, Federal Republic of	LU	Luxembourg	US	United States of America
DK	Denmark	MC	Monaco		

MACHINE PROCESS FOR TRANSLATING PROGRAMS IN BINARY MACHINE LANGUAGE INTO ANOTHER BINARY MACHINE LANGUAGE

Background of the Invention

This invention relates to machine processes for translating computer programs from one computer language into another computer language and, more particularly, to a method for translating computer programs from one binary machine language into another binary machine language, or from one assembly language into another assembly language.

Description of the Prior Art

The art relating to machine processes for translating computer programs from one computer language to another ("translators") is well-developed with an extensive amount of literature. The following text briefly describes the relevant technologies.

Compilers are well-known in the art. They translate programs written in a high-level language, such as C, Fortran, or Pascal, into either assembly language or binary machine language. Likewise, assemblers are well-known in the art; they translate assembly language into binary machine language.

In general, compilers translate single lines of a human-readable, high-level language ("statements") into several lines of assembly language or several binary machine instructions. Assemblers, on the other hand generally translate one assembly language line into one

machine instruction (neglecting comments and assembler directives).

There is, therefore, scope for optimization with compilers that is not present with assemblers. A good, i.e., optimizing, compiler is usually one that generates fewer machine instructions for a particular sequence of statements than does an average compiler. A vast array of techniques, well-known in the art, have been developed to optimize compiler code generation, including "global flow analysis". The standard reference for compiler design is Compilers, Principles, Techniques, and Tools by Aho, R. Sethi, and J. Ullman (Addison-Wesley, 1986); note especially Chapter 10 on optimization techniques.

Interpreters are similar to compilers, but instead of simply translating the source program into machine language, an interpreter translates each statement, then executes the translated code, and then translates and executes the next statement, and so on. Because an interpreter deals with only one statement at a time, it can be simpler in design than a compiler for the same language, but the scope for optimization is much less. Consequently, interpreted programs tend to execute much more slowly than compiled programs.

Other forms of translators have also been developed from time to time. Various high-level translators (e.g., Pascal to C) have been around for almost as long as there have been high-level languages. Assembly language translators (e.g., 8080 assembly code to 8086 assembly code) have also been reported, but seldom actually seen. It appears that compiler optimization techniques have not been applied to such translators.

Considering now binary machine language source files, disassemblers have been standard features of

debugging tools for years and are well-known in the art. They translate sections of binary machine language into an equivalent set of assembly language instructions. Their use has been restricted because of several well-known problems, in particular the "phase problem" and the "data problem".

The phase problem is caused because the binary instruction formats of most computers have a varying length. It is, therefore, sometimes difficult to know where one instruction ends and another begins. It is especially difficult to know whether the disassembly process has been started correctly at the beginning of an instruction, or has begun in the middle of an instruction. In the latter case, all subsequent disassembled instructions will generally be wrong.

The data problem arises because many programs contain bytes or words of data interspersed with the instructions.

Disassemblers have a difficult time determining whether a particular pattern of bits is really an instruction or just several bytes of data. And of course the data problem exacerbates the phase problem, since the disassembler must correctly determine the length of the data area before it can resume disassembling at the correct place.

Like disassemblers, simulators deal with binary machine language source files. Simulators, however, are similar to interpreters in that they simultaneously translate and execute the source file. When they run, they have an effect as if the source binary program were executing on another computer with a different machine language. Simulators achieve this effect by simulating in software on the target computer the exact behavior of the original computer. Simulators have had very limited success, largely because of one well-known

problem: they are very slow. Quite often hundreds of simulator instructions must be executed for every instruction in the source program, and even the very best simulators need ten to twenty simulator instructions per source instruction. Because of the above-described problems with disassemblers, there appear to be no example of binary-to-binary optimizing translators; that is, of programs that translate one binary machine language into another binary machine language with high efficiency. Thus, there are no binary-to-binary equivalents of compilers, as simulators are binary-to-binary equivalents of interpreters.

Summary of the Invention

This invention provides an effective machine process for translating with high efficiency computer programs from one binary machine language into another binary machine language. This machine process will be referred to as a "binary compiler"; it can be realized in a program for a digital computer. The technique can also be used to translate from one assembly language into another. A binary compiler has the same relationship to a simulator that a compiler has to an interpreter. And just as a compiler produces code that executes faster than interpreted code, application programs converted with a binary compiler execute faster than they do with simulators. The binary compiling process of the present invention involves disassembling the source binary program, analyzing the binary program to produce "global flow analysis" data, using this global flow analysis data to complete the disassembly process, and producing a translated binary machine language version of the source binary program, using global flow analysis data to generate optimized

binary code. When the invention is used to translate from one assembly language into another, the disassembly stage is omitted, but the global flow analysis is still performed. The output uses the global flow analysis data to produce optimized assembly code instead of optimized binary code.

Brief Description of the Drawings and Listings

A complete understanding of the present invention and of the above and other advantages thereof may be gained from a consideration of the following detailed description of an illustrative embodiment thereof, which translates programs from the binary machine language of the Intel 8086 microprocessor ("8086 code") into the binary machine language of the Motorola 68020 microprocessor ("68020 code").

FIG. 1 shows a generalized flow diagram representing the illustrative machine algorithm for practicing data processing in accordance with the present invention;

FIG. 2 shows a more detailed flow diagram of the PROCESS PROCEDURES portion of the algorithm of FIG. 1;

FIG. 3 shows a more detailed flow diagram of the PROCESS A PROCEDURE portion of the algorithm of FIG. 2;

FIG. 4 shows a more detailed flow diagram of the of the BUILD BASIC BLOCKS portion of the algorithm of FIG. 3;

FIG. 5 shows a more detailed flow diagram of the FORWARD FLOW ANALYSIS portion of the algorithm of FIG. 3;

FIG. 6 shows a more detailed flow diagram of the BACKWARD FLOW ANALYSIS portion of the algorithm of FIG. 3;

FIG. 7 shows a more detailed flow diagram of the UNKNOWNS ANALYSIS portion of the algorithm of FIG. 3;

FIG. 8 shows a more detailed flow diagram of the ANALYZE COMPLETED FLOWGRAPH portion of the algorithm of FIG. 1;

FIG. 9 shows a more detailed flow diagram of the LIVE/DEAD ANALYSIS portion of the algorithm of FIG. 8;

FIG. 10 shows a more detailed flow diagram of the TRANSLATE INSTRUCTIONS portion of the algorithm of FIG. 1.

Detailed Description

The first steps of the algorithm represented by the flow chart of FIG. 1 are to read the input data 104 necessary to carry out the represented process. This data includes the source program 108 in 8086 binary machine language and any application specific data (asd) 112 associated with the source binary program. Read data input is represented by block 104 of FIG. 1.

Following input of data, the process enters the PROCESS PROCEDURES algorithm block 120 of FIG. 1. The object in this PROCESS PROCEDURES 120 is to analyze the source binary program into its component instructions, which are grouped into "basic blocks" of sequential instructions terminated by a change of control (call, jump, or return). The PROCESS PROCEDURES 120 also builds a "flow graph", which is a data structure representing the flow of control among the basic blocks. Associated with each basic block are data structures containing information about the use of registers, flags, the stack, and memory within the block, along with a list of all instructions in the block. The basic blocks are grouped into "procedures", which are entered from a call instruction and terminate with a return.

After the PROCESS PROCEDURES algorithm 120 has been run, the process enters the ANALYZE COMPLETED FLOWGRAPH algorithm represented by block 124 of FIG. 1. The

object in the ANALYZE COMPLETED FLOWGRAPH 124 is to analyze the data structures built by PROCESS PROCEDURES 120 in several different ways and pass the results of this analysis to the TRANSLATE INSTRUCTIONS block 132. The ANALYZE COMPLETE FLOWGRAPH 124 performs five different types of analysis: "call-return analysis", "live-dead analysis" on registers, "live-dead analysis" on flags, "byte orientation analysis", and "alignment analysis". The results of this analysis are used by TRANSLATE INSTRUCTIONS 132 to generate optimized translated code.

After the ANALYZE COMPLETED FLOWGRAPH algorithm 124 has been run, the process enters the TRANSLATE INSTRUCTIONS algorithm block 132. The object in the TRANSLATE INSTRUCTIONS 132 is to translate the analyzed instructions in the instruction lists of the basic blocks into equivalent instructions in 68020 binary machine language, using the data developed by the ANALYZE COMPLETED FLOWGRAPH 124 to make the translated code sequences optimally short. The result of applying TRANSLATE INSTRUCTIONS 132 is to produce the translated binary program 140. The PROCESS PROCEDURES 120, ANALYZE COMPLETED FLOWGRAPH 124, and TRANSLATE INSTRUCTIONS 132 algorithms of FIG. 1 will now be described in detail.

FIG. 2 shows an overview of the PROCESS PROCEDURES process 120. During the course of its operation, the process manipulates data structures called Procedure Blocks (PBs), which can be linked onto up to three different queues: the new procedure queue, the upward procedure queue, and the downward procedure queue. One PB exists for every procedure in the program being analyzed.

As indicated in FIG. 2, the first step of the PROCESS PROCEDURES process 120 is to build an empty PB on the new procedure queue. This step is represented by

block 204 of FIG. 2. The next step, represented by block 208, is to determine whether any PBs exist on the new procedure queue. (The first time through the loop the answer is, of course, yes.) If a PB is on the queue, the process moves to block 210, which is responsible for removing the empty PB from the new procedure queue and initializing it, that is, filling in the starting values in the PB. After the step represented by block 210, the process moves to block 220, which represents the PROCESS A PROCEDURE algorithm. This algorithm 220 performs as much processing on the procedure represented by the PB as can be done at the current point in the analysis. As many basic blocks are built as can be found in this procedure. One result of this algorithm's operation may be to cause new PBs to be put on the new procedure queue, or existing PBs (including the present one) to be put on either the downward procedure queue or the upward procedure queue.

After this algorithm completes, the process moves back to the decision step represented by block 208. This step determines again if any PBs exist on the new procedure queue (some may have been created by the PROCESS A PROCEDURE algorithm). Again, the process moves to block 210 and thence to 220 and back to 208 if a PB is found on the queue, with this loop executing until no new PBs remain on the new procedure queue.

At this point, the process moves on to the decision step represented by block 212, which determines if any PBs are on the downward procedure queue. If any are found, the process moves to block 214, which removes the PB from the queue for processing, and on to block 220 (PROCESS A PROCEDURE). The result of this step may be to create new PBs, so the process moves back to 208, and this cycle continues until all PBs have been removed

from both the new procedure queue and the downward procedure queue.

Then the process moves to the step represented by block 216 and decides if there are any PBs on the upward procedure queue. If there are, they are processed just like the PBs on the downward queue, until no PBs remain on any of the three queues, then the whole PROCESS PROCEDURES algorithm exits.

FIG. 3 shows the details of the PROCESS A PROCEDURE block 220 shown in FIG. 2. During the course of these steps, the binary compiler manipulates four queues of data structures called Basic Blocks (BBs). As will be discussed, the four queues are: new queue, config queue, unknowns queue, and uses queue. One BB is associated with each basic block in the procedure. As indicated in FIG. 3. the first step in this process is to build all BBs that can be identified in the procedure (304). Then FORWARD FLOW ANALYSIS is performed (block 308), then BACKWARD FLOW ANALYSIS (312), and finally UNKNOWNS ANALYSIS (320). If any BBs remain on any of the four queues after these steps (see block 3200, then the process repeats steps 304 through 316, and this loop is iterated until no BBs remain on any queues.

Then the process tests to see if any of the backward flow data for the procedure's own PB has been modified (324). If it has been modified, then the PBs for all procedures that call this procedure are put on the upward procedure queue which would be processed subsequently by blocks 216 and 218 shown in FIG. 2.

FIG. 4 shows the details of the BUILD BASIC BLOCKS 304 algorithm of FIG. 3, which builds all BBs that can be identified in the procedure. The first step (402) is to take the first available BB off the new queue (one of the four BB queues mentioned above), and then sequentially disassemble instructions beginning at the

starting address in the BB, until a jump, call, return, or interrupt instruction (known as the termination instruction) is reached. An encoded representation of the disassembled instructions is stored in a data structure called the Instruction List associated with the BB.

The next step (404) is to link the current BB with the BB whose code begins immediately after the current BB's termination instruction (this BB is called the immediate successor). If no immediate successor BB has yet been created, the process now creates a new BB with this address, links it to the current BB, and also puts it on the new queue. If an immediate successor already exists, it is put on the config queue (another of the four BB queues).

The next step (408) is to perform forward flow analysis within the current BB. This means to take the forward data (register values, stack values, flag values, memory values) stored in all BBs that logically precede the current BB and propagate these values through the BB to its end, performing all transformations on the data that are done by the BB's instructions. The BBs that precede the current BB are all called its predecessors; they are either the immediate predecessors in the sense of the previous paragraph, or they are BBs whose termination instruction resulted in a transfer of control to the current BB. The result of this operation is the forward data of the current BB, which is stored in a data structure associated with the BB.

The next steps depend on whether the termination instruction of the current BB is a call, a computed jump, or an interrupt (block 412). If it is (420), then the BB is put on the unknowns queue (another of the four BB queues). If it is not (i.e., if the termination

instruction is a simple jump), then the BB is linked to the BB associated with the code at the jump's target address. A new BB is created and put on the new queue, if no BB exists for that address. An existing BB is put on the config queue. (See 416).

Finally, in all cases (424), the current BB is put on the uses queue (the last of the four BB queues). Then a check is made to see if the new queue is empty (428). If it is not, the process performs steps 402 through 428 again repeatedly until the new queue is empty.

FIG. 5 shows the details of the FORWARD FLOW ANALYSIS algorithm of FIG. 3, represented by block 308, which performs the forward flow analysis on all the BBs in a procedure. The first step (504) is to take a BB off the config queue. Then the forward data from all predecessors to the BB is propagated through the BB and stored in its forward data structure (blocks 508 and 512). If the immediate predecessor ends with a call instruction, the data is first propagated through the called procedure before it is propagated through the BB. Then a check is made (516) to determine if the current BB's forward data has been modified. If so, all successor BBs are put on the config queue (520). In either case, a check is made to see if any BBs are left on the config queue (524), and if there are, the whole loop from 504 to 524 is performed again repeatedly until no BBs remain on the config queue.

FIG. 6 shows the details of the BACKWARD FLOW ANALYSIS algorithm of FIG. 3, represented by block 312, which performs the backward flow analysis on all the BBs in a procedure. The first step (604) is to take a BB off the uses queue. Then the backward data (expression lists) from all successors to the BB is propagated back through the BB and stored in its backward data structure

(blocks 608 and 612). If any successor ends in a call instruction, the data is first propagated through the called procedure before it is propagated through the BB. Then a check is made (616) to determine if the current BB's backward data has been modified. If so, all predecessor BBs are put on the uses queue (620). In either case, a check is made to see if any BBs are left on the uses queue (624), and if there are, the whole loop from 604 to 624 is performed again repeatedly until no BBs remain on the uses queue.

FIG. 7 shows the details of the UNKNOWNS ANALYSIS algorithm of FIG. 3, represented by block 316, which performs the analysis on the unknown BBs in a procedure. The first step (704) is to take a BB off the unknown queue. Then an attempt is made to calculate the computed jump or call address, using data in the asd file, if necessary (706). The process then splits (712) depending on the result of this attempt. If the computation successfully determined the target address, a link is built to the target BB. A new BB is created on the new queue, if no BB exists for that address; an existing BB is put on the config queue. (See block 724.) Then the BB itself is put on the uses queue, and that entry is removed form the unknown list of the procedure's PB (724). If the attempt to compute the target address was unsuccessful (720), the BB is put back on the unknown queue and an entry is made in the unknown list of the procedure's PB. In either case, a check (728) is made to see any BBs remain on the unknown queue whose unknown list entry has not been processed. If any remain, the entire loop from 704 to 728 is performed again repeatedly until no unprocessed BBs remain on the unknowns queue.

FIG. 8 shows the details of the ANALYZE COMPLETED FLOWGRAPH algorithm of the FIG. 1 process, represented

by block 124, which performs the global flow analysis on the completed flowgraph in preparation for the code generation stage. The first step 804 is the sorting stage where the complete set of basic blocks are sorted into order by increasing address, using a standard sort algorithm. In this particular implementation, the sort algorithm used is a version of the insertion sort algorithm described in D.E. Knuth The Art of Computer Programming, Vol. 3., Sorting and Searching, Addison-Wesley, 1973, Reading, Massachusetts, pages 80 - 102. After completion of the sorting stage, the process moves to an optional stage (808) that optimizes the use of jump, call, and return addresses for the particular case of translating programs from 8086 code into 68020 code. This stage will not be present in the general case.

After the step represented by block 808, the process moves to block 812, which represents the LIVE-DEAD ANALYSIS algorithm. This step is the heart of the optimizations performed by the ANALYZE COMPLETED FLOWGRAPH algorithm 124. This step performs a global flow analysis of the completed flowgraph, computing "live-dead" data for registers and flags. This data specifies whether source machine condition flags (e.g., CARRY or OVERFLOW) and registers are used by subsequent instructions ("live") or not used ("dead"). This information is used by the subsequent TRANSLATE INSTRUCTIONS algorithm 132 in FIG. 1 to generate optimized target code, by only generating instructions to preserve or simulate live condition flag values and by not preserving data in dead registers. Then the process moves to another optional stage (816), which performs various "peep-hole" optimizations that are dependent on the particular case of translating programs from 8086 code into 68020 code.

FIG. 9 shows the details of the stage represented by block 812 in FIG. 8., that is, the LIVE-DEAD ANALYSIS algorithm. During the course of these steps two queues of Procedure Blocks (PBs) -- the upward queue and the downward queue -- and one queue of Basic Blocks (BBs) are completed. The first step (904) is to put the PBs for all non-returning procedures (that is procedures that do not return to another procedure) on the downward queue. The next step (908) is to check whether there are any PBs on the downward queue. If not, the process checks (912) to see if there are any PBs on the upward queue. If there are any PBs on either queue, the process continues to step 916 which computes live-dead data and the transfer function (that is, the effect the procedure has on live-dead data coming from other procedures) for the procedure. This step terminates when the procedure is completely analyzed or when the process encounters a call instruction whose called procedure has not been previously analyzed.

Then the process continues to step 920, which determines whether step 916 terminated because the procedure was completely analyzed or because a call to an unanalyzed procedure was reached. If the procedure was completed, the process moves to step 924, which puts the PBs for all procedures that call the completed one on the upward queue. If step 916 terminated because a call to an unanalyzed procedure was reached, the unanalyzed procedure's PB is put on the downward queue in step 928. In either case, the process continues back to step 908 and iterates until all procedures have been fully analyzed.

FIG. 10 shows the details of the TRANSLATE INSTRUCTIONS algorithm of the FIG. 1 process, represented by block 132, which performs the 68020 code generation. The first step (1204) is to get a BB from

the chain of BBs arranged in increasing address order that is produced by step 804. Then, an instruction's encoded opcode, addressing mode, and attributes are read from the instruction list associated with the BB (1208). These encoded values are used as indices to select a short segment of translated 68020 binary code (often just one instruction) from a table (1212). The 68020 code is saved in a data structure (1216), then the process proceeds to convert the next instruction by performing steps 1208 through 1220 repeatedly until no more instructions remain in the BB's instruction list.

At this point the 68020 instructions representing the BB are output to a file (1222). Then the process moves to the next BB and performs steps 1204 through 1224 until no BBs remain to be translated.

It is to be understood that the above-described embodiments and program implementations are only illustrative of the application of the principles of the present invention. Numerous modifications may be devised by those skilled in the art without departing from the spirit and scope of the invention.

WHAT IS CLAIMED IS:

1. A machine process for translating a first computer program in one binary machine language having one or more basic blocks, into a second computer program in another binary machine language, by use of a programmed digital computer, having stored in its internal memory a program enabling the computer to perform the following steps:

a. disassembling one of said basic blocks of said first computer program;

b. analyzing one basic block to produce global flow analysis data;

c. continuing the steps of (a) and (b) until all of said basic blocks of said first computer program have been disassembled and analyzed; and

d. generating said second computer program, using said global flow analysis data.

2. The process of Claim 1 wherein the disassembling step further comprising:

a. disassembling the instructions of said one basic block continuously until a branch, call, or return instruction is reached; and

b. preserving a representation of the opcode and addressing modes of each disassembled instruction.

3. The process of Claim 2 wherein said analyzing step further comprising:

a. computing and saving the target address of the branch, call, or return instruction; and

b. saving the next sequential address an unconditional branch instruction.

4. The process of Claim 3 wherein said continuing step further comprising:

a. continuing disassembling at a saved address (either a target address of a branch, call, or return, or the next sequential address after a conditional branch); and

b. performing the steps in Claims 2-3 repeatedly until either step (a) of this claim encounters only code that has already been disassembled, or the source program ends.

5. The process of Claim 4 wherein said generating step further comprising:

a. sorting the sections of disassembled code into order by their address; and

b. generating the binary machine language translation for each disassembled instruction.

6. The process of Claim 3 wherein the computing step further comprises

a. continuously computing and updating the currently known contents of the first computer program's stacks to produce "global flow analysis stack data"; and

b. using said global flow analysis stack data to compute target addresses for branches, calls, and returns.

7. The process of Claim 3 wherein the computing step further comprises:

a. continuously computing and updating the program's registers to produce "global of low analysis register data"; and

b. using said global flow analysis register

data to computer target addresses for branches, calls and returns.

8. The process of Claim 3 wherein the computing step further comprises:

a. continuously computing and updating the currently known contents of the first computer program's key memory location to produce "global flow analysis memory data"; and

b. using said global flow analysis memory data to computer target addresses for branches, calls and returns.

9. The process of Claim 3 wherein the computing step further comprises:

a. using application specific data to compute branches, calls or return target addresses that cannot be computed by the method of Claims 6, 7, or 8.

10. A process as in Claim 9 wherein said application specific data is used to determine the contents of the stack, registers, or memory locations.

11. The process of Claim 5 wherein said global flow analysis data is used to compute whether source machine condition flags (e.g., CARRY or OVERFLOW) are used by subsequent instructions ("live") or not used ("dead"), and this information is then used to generate optimized target code, by only generating instructions to preserve or simulate live condition flag values.

12. The process of Claim 11 wherein said global flow analysis data is used to compute whether source machine registers are used by subsequent instructions

("live") or not used ("dead"), and this information is then used to generate optimized code, by not preserving data in dead registers.

13. The process of Claim 5 wherein said global flow analysis data (stack, register, or memory) is used to compute whether source machine registers are used by subsequent instructions ("live") or not used ("dead"), and this information is then used to generate optimized code, by using the target machine equivalents of dead registers to hold temporary variables needed by the translated instructions.

14. The process as in Claims 5 wherein said global flow analysis data is used to compute whether data in memory of the source machine is referenced by two or more instructions that operate on different data type lengths (e.g., 16 bit quantities and 8 bit quantities), and this information is used to generate optimized code, in cases where the target machine language and the source machine language differ in the ordering of bytes within a half-word, and half-words within a word.

15. The process of Claim 5 wherein said global flow analysis data is used to compute whether data in memory of the source machine is aligned on addresses that are a multiple of the data length (e.g., 2 bytes or 4 bytes). Then for those target machines that require such alignment, generating a single memory reference to access the data if the data is aligned, and generating multiple references only when the data is not aligned or when its alignment cannot be determined.

16. The process of Claim 7 wherein said global flow analysis register data is used to identify

operating system calls embedded in the binary machine language, by virtue of the register contents current when the call is made.

17. A machine process for translating a first computer program in a computer assembly language having an entry point into second computer assembly language by use of a programmed digital computer having stored in its internal memory a program enabling the computer to perform the following steps:

a. starting at said entry point, computing global flow analysis data (stack, register, and/or memory) continuously until the first program ends;

b. using said global flow analysis data (stack, register, or memory) for computing whether the first program condition flags (e.g., CARRY or OVERFLOW) are used by subsequent instructions ("live") or not used ("dead"); and

c. generating optimized assembly code using this information, by only generating instructions to preserve or simulate live condition flag values.

18. The process as in Claim 17 wherein said step (a) further comprising:

a. reading the first program in address order,

b. starting at the entry point, computing global flow analysis data (stack, register, and/or memory) continuously until a branch, call or return instruction is reached,

c. computing and saving the target address of the branch, call, or return instruction,

d. saving the next sequential address an unconditional branch instruction,

e. continuing computing global flow analysis data at a saved address (either a target address of a branch, call, or return, or the nest sequential address after a conditional branch),

f. performing steps (b) through (e) of this claim repeatedly until either step (e) encounters only code that has already been completely analyzed, or the source program ends.

19. A machine process for translating a first computer program in a computer assembly language having an entry point into second computer assembly language by use of a programmed digital computer having stored in its internal memory a program enabling the computer to perform the following steps:

a. starting at said entry point, computing global flow analysis data (stack, register, and/or memory) continuously until the first program ends;

b. using said global flow analysis data (stack, register, or memory) for computing whether the first program registers are used by subsequent instructions ("live") or not used ("dead"); and

c. generating optimized assembly code using this information, by not preserving data in dead registers.

20. A machine process for translating a first computer program in a computer assembly language having an entry point into second computer assembly language by use of a programmed digital computer having stored in its internal memory a program enabling the computer to perform the following steps:

a. starting at said entry point, computing global flow analysis data (stack, register, and/or memory) continuously until the first program ends;

b. using said global flow analysis data (stack, register, or memory) for computing whether the first program registers are used by subsequent instructions ("live") or not used ("dead"); and

c. generating optimized assembly code using this information, by using the second program's equivalents of dead registers to hold temporary variables needed by the translated instructions.

21. A machine process for translating a first computer program in a computer assembly language having an entry point into second computer assembly language by use of a programmed digital computer having stored in its internal memory a program enabling the computer to perform the following steps:

a. starting at said entry point, computing global flow analysis data (stack, register, and/or memory) continuously until the first program ends;

b. using said global flow analysis data (stack, register, or memory) for computing whether data in memory of the source machine is referenced by two or more instructions that operate on different data type lengths (e.g., 16 bit quantities and 8 bit quantities); and

c. generating optimized code using this information, in cases where the second program's machine language and the first program language differ in the ordering of bytes within a half-word, and half-words within a word.

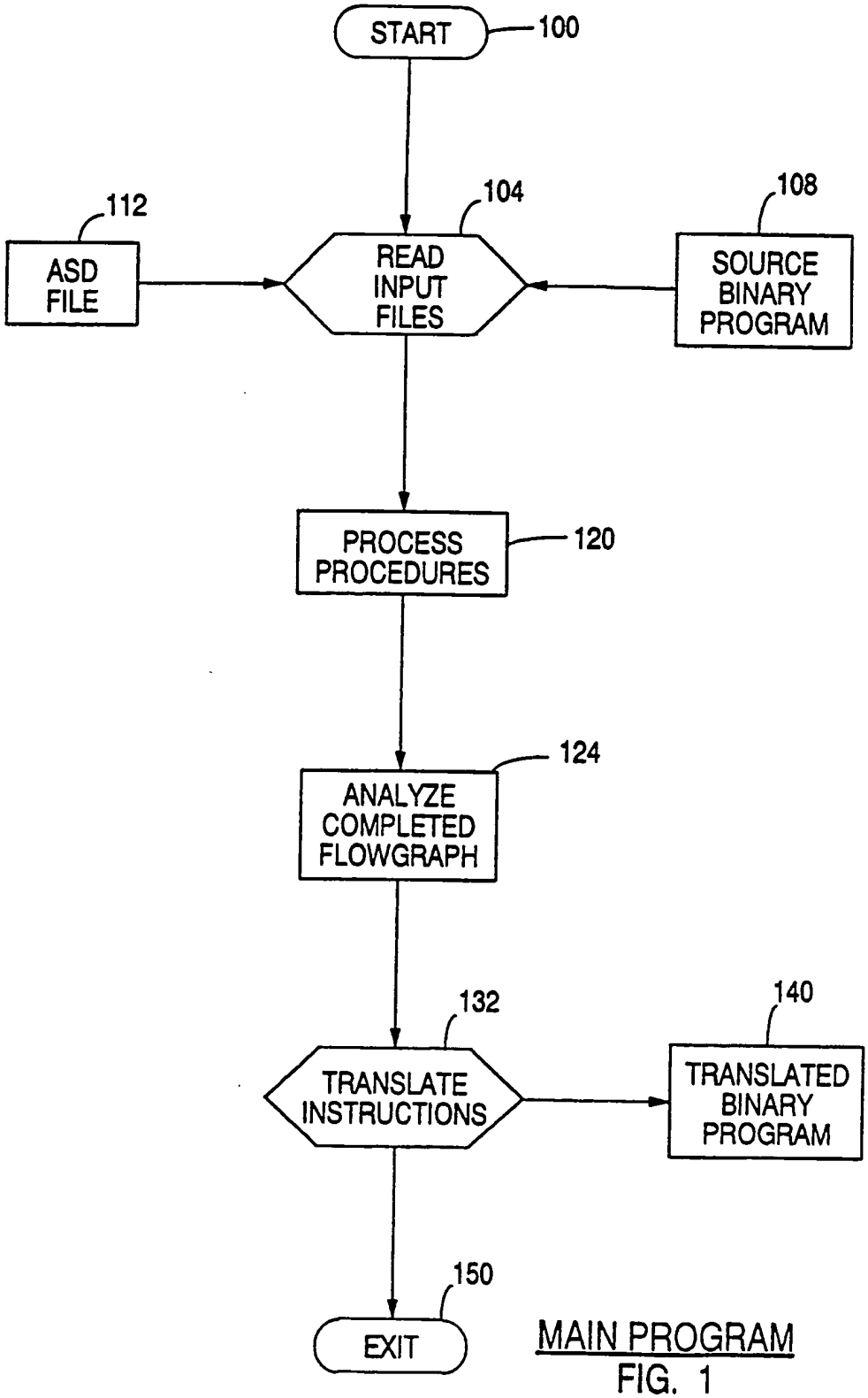
22. A machine process for translating a first computer program in a computer assembly language having an entry point into second computer assembly language by use of a programmed digital computer having stored in

its internal memory a program enabling the computer to perform the following steps:

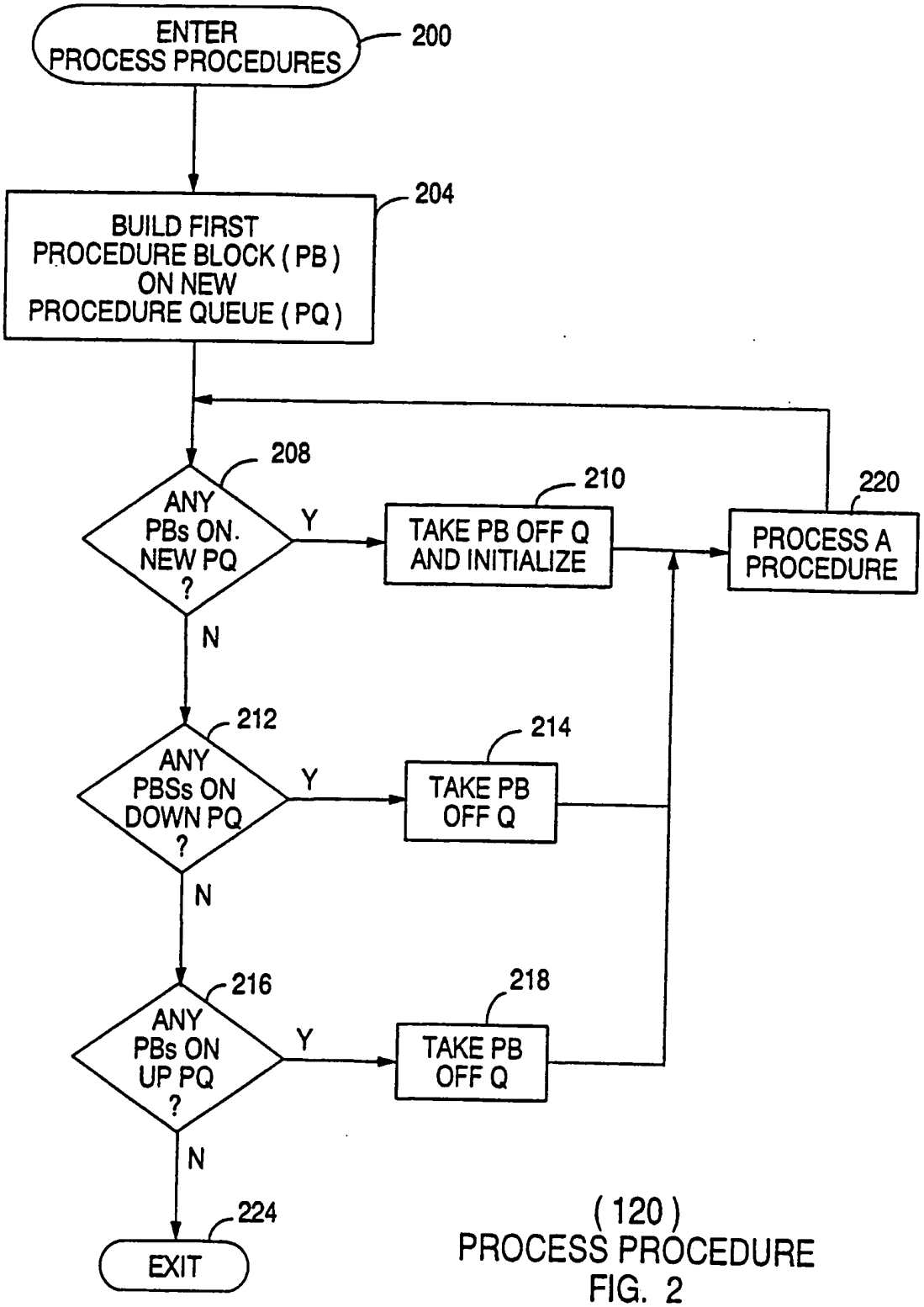
a. starting at said entry point, computing global flow analysis data (stack, register, and/or memory) continuously until the first program ends;

b. using said global flow analysis data (stack, register, or memory) for computing whether data in memory of the source machine is aligned on addresses that are a multiple of the data length (e.g., 2 bytes or 4 bytes); and

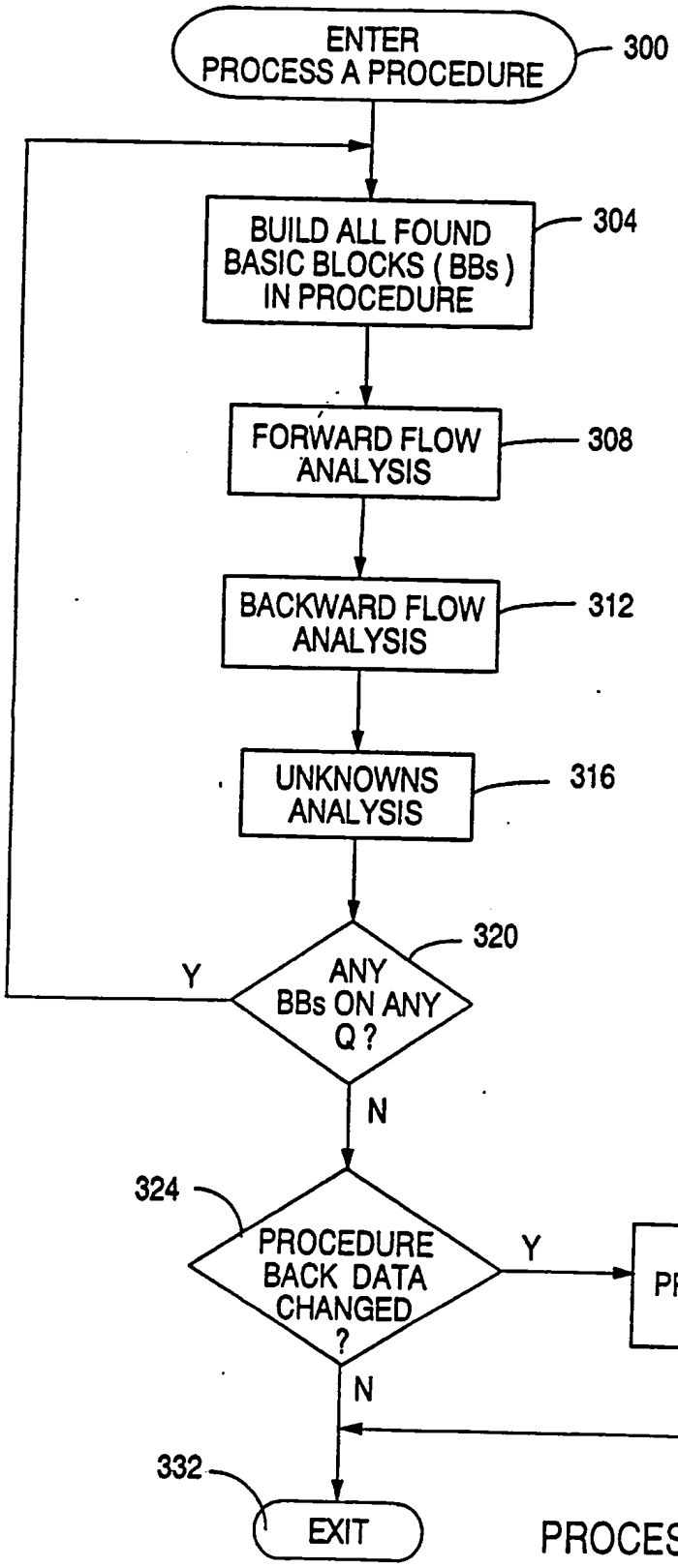
c. generating a single memory reference to access the data if the data is aligned, and generating multiple references only when the data is not aligned or when its alignment cannot be determined for those second language that require such alignment.



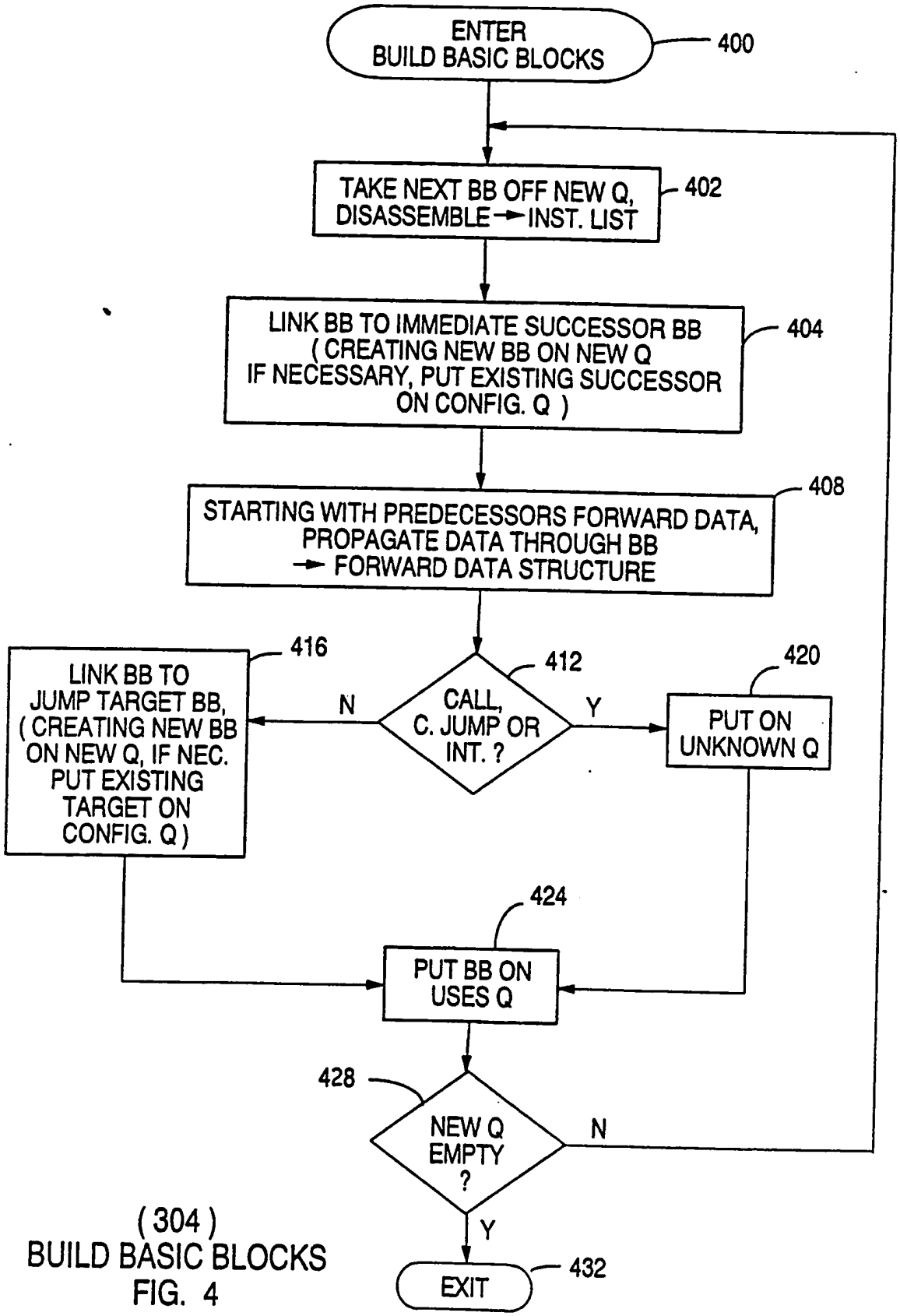
MAIN PROGRAM
FIG. 1



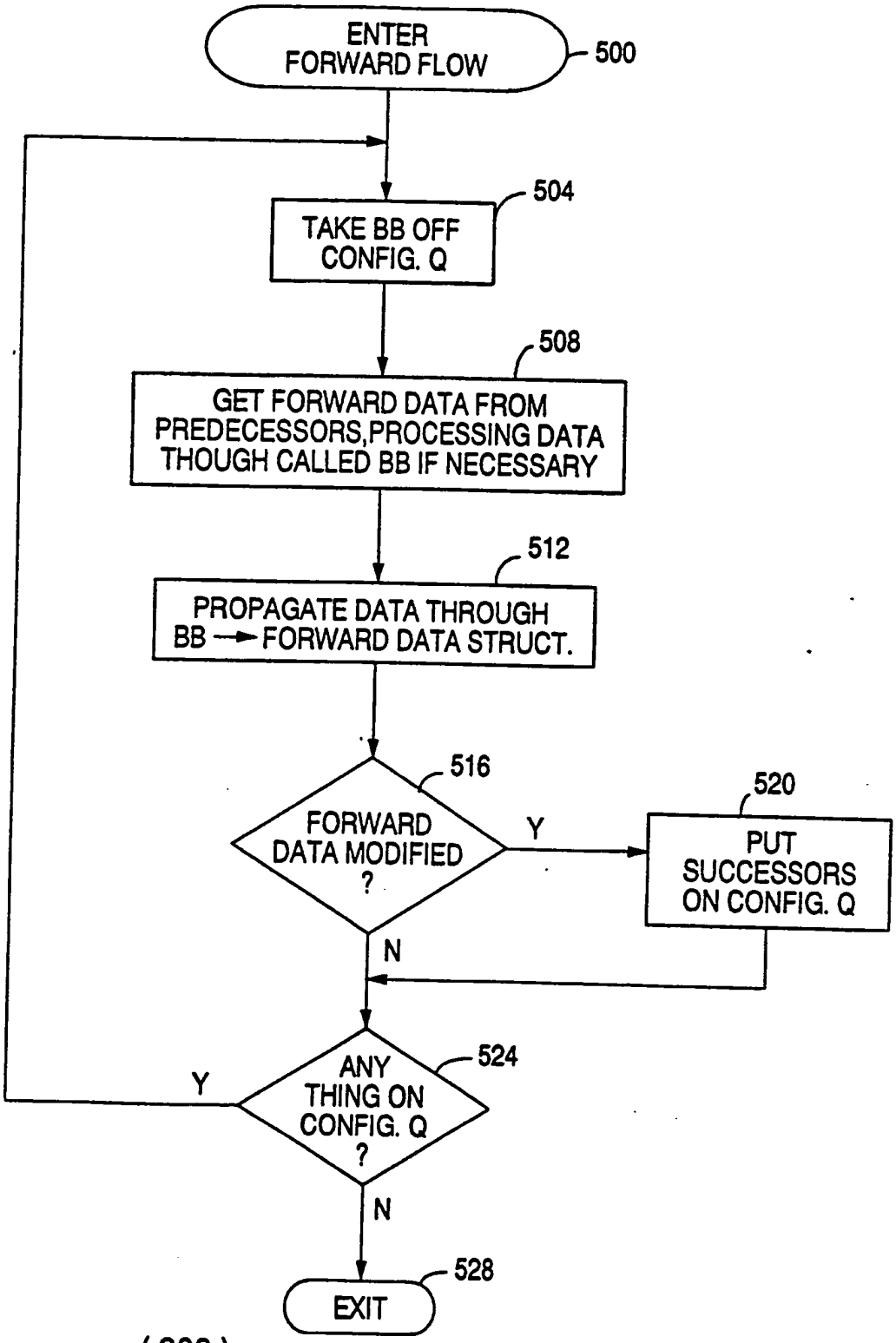
(120)
PROCESS PROCEDURE
FIG. 2



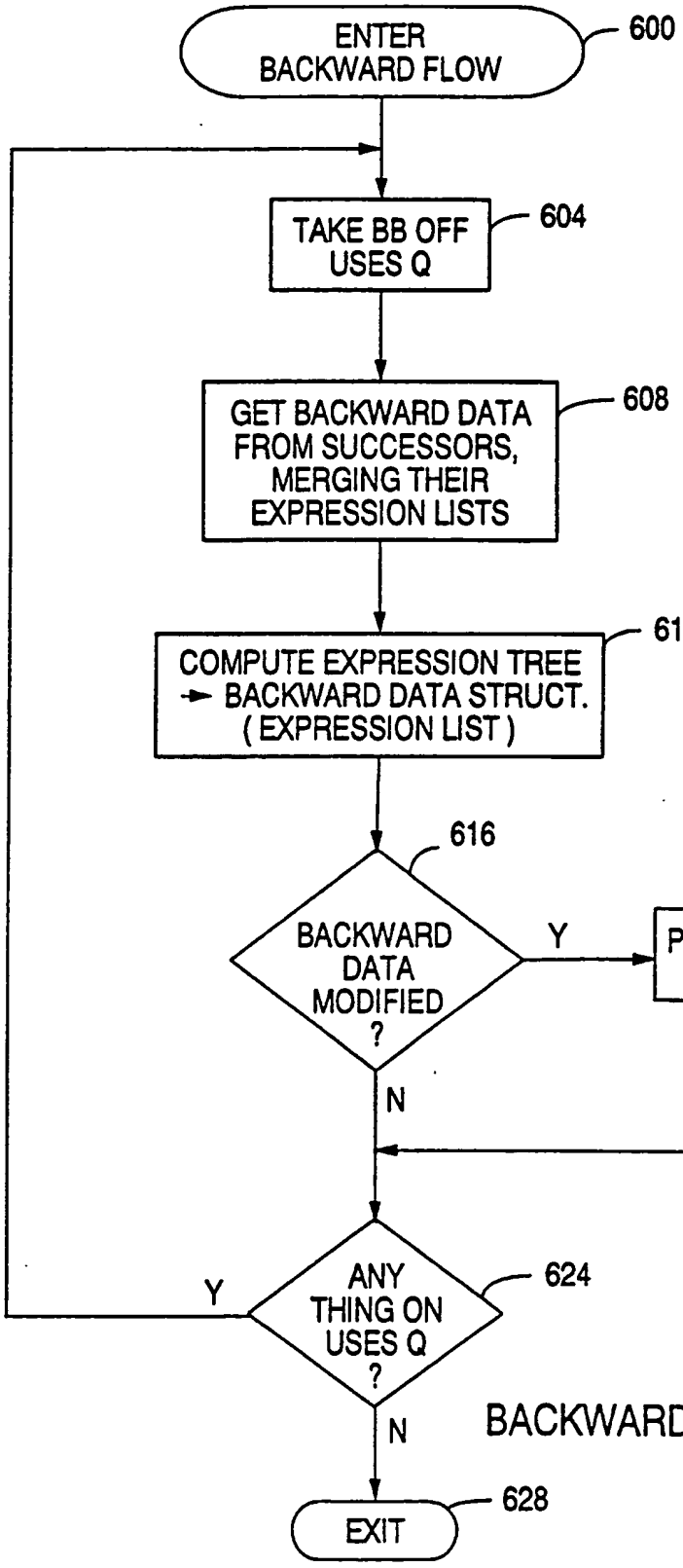
(220)
PROCESS A PROCEDURE
FIG. 3



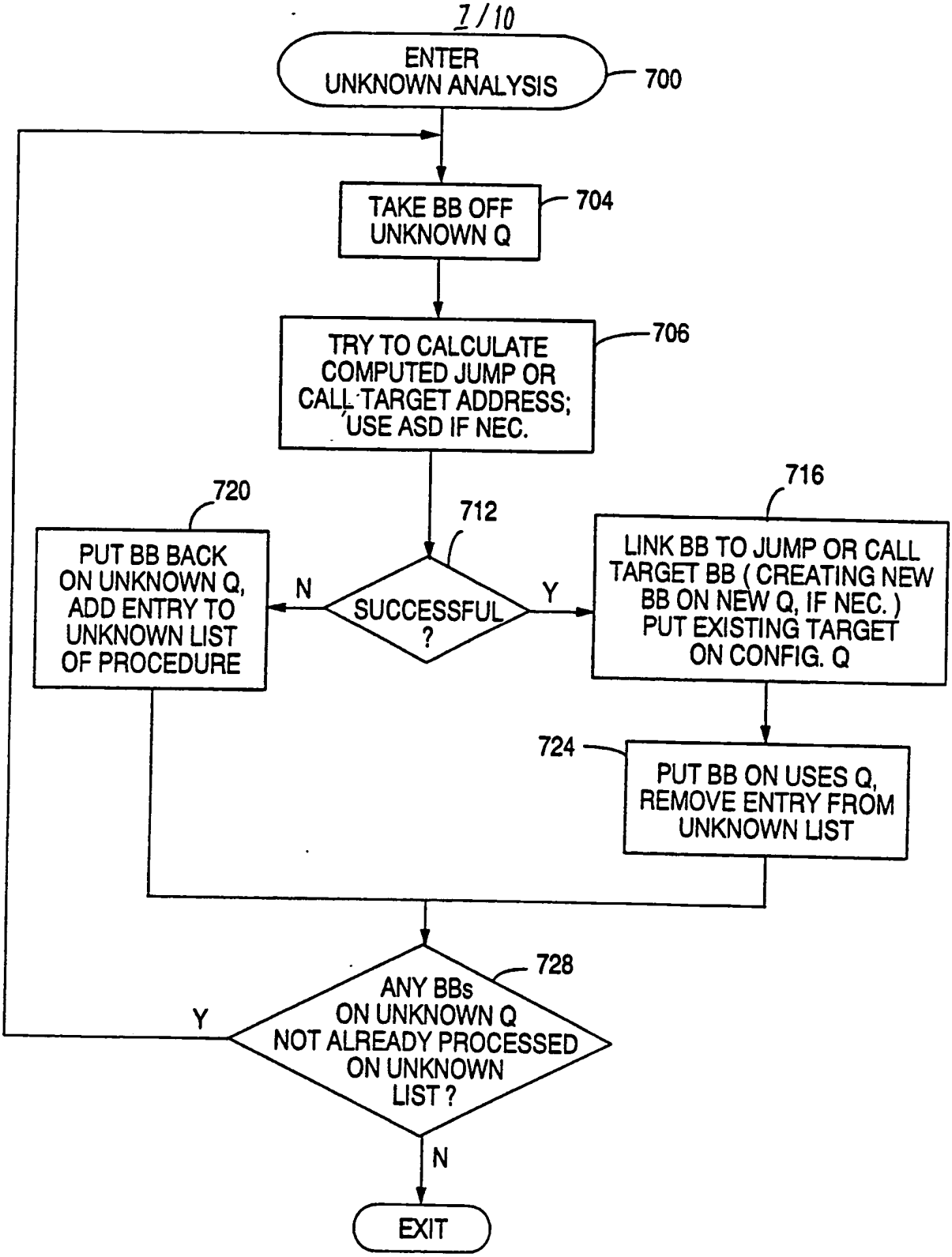
(304)
BUILD BASIC BLOCKS
FIG. 4



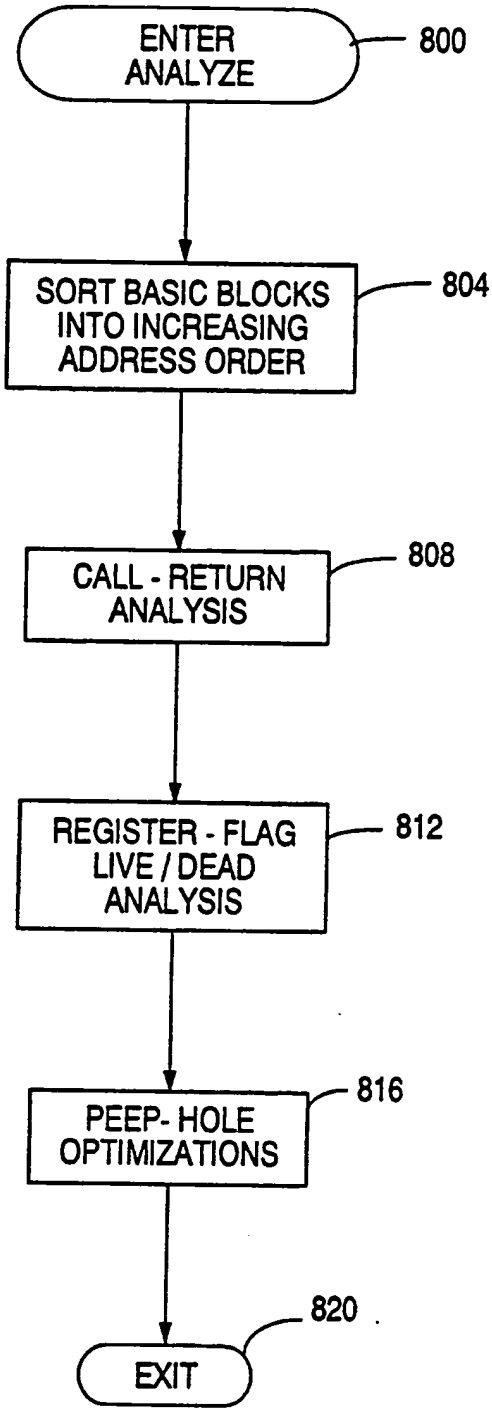
(308)
FORWARD FLOW ANALYSIS
FIG. 5



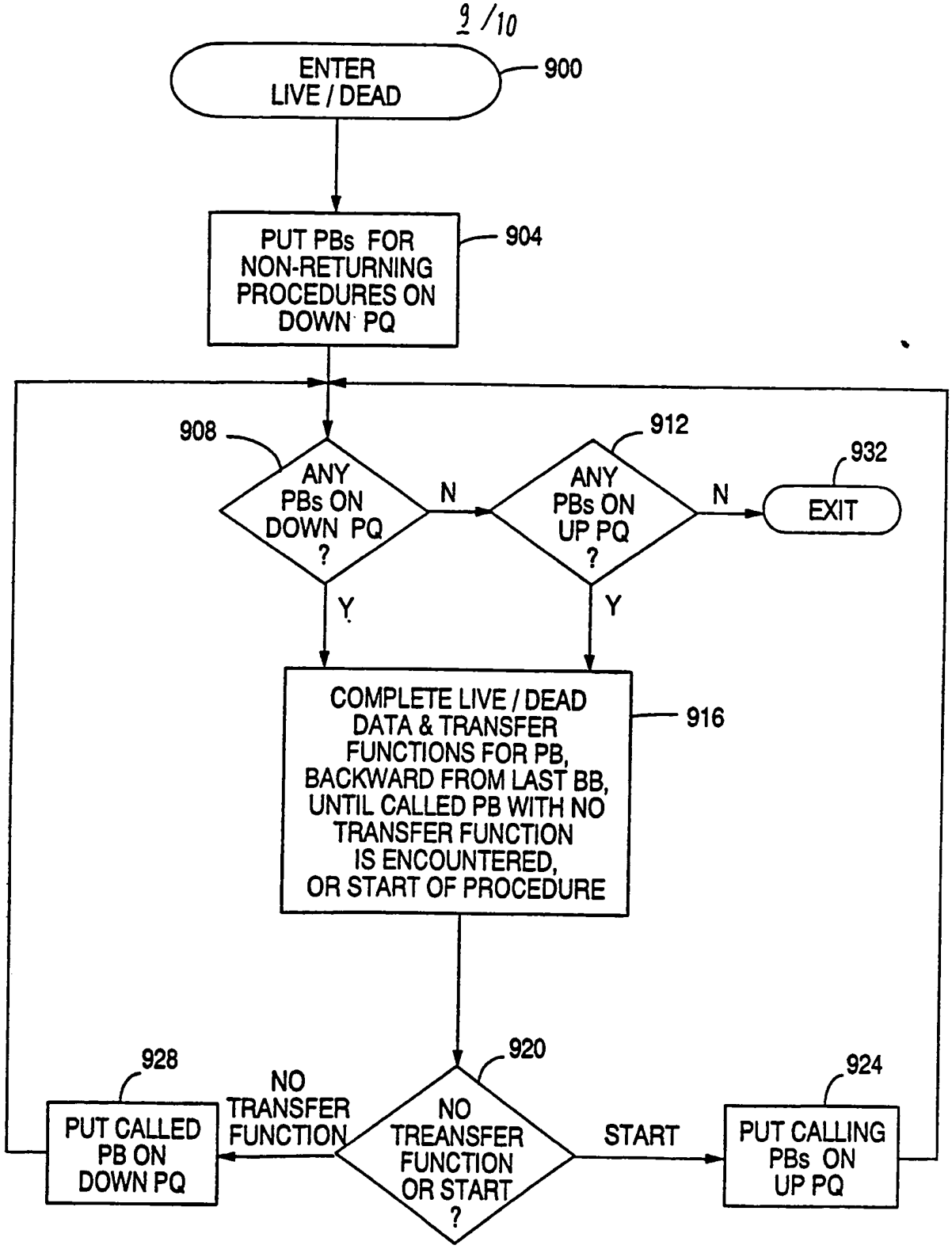
(312)
BACKWARD FLOW ANALYSIS
FIG. 6



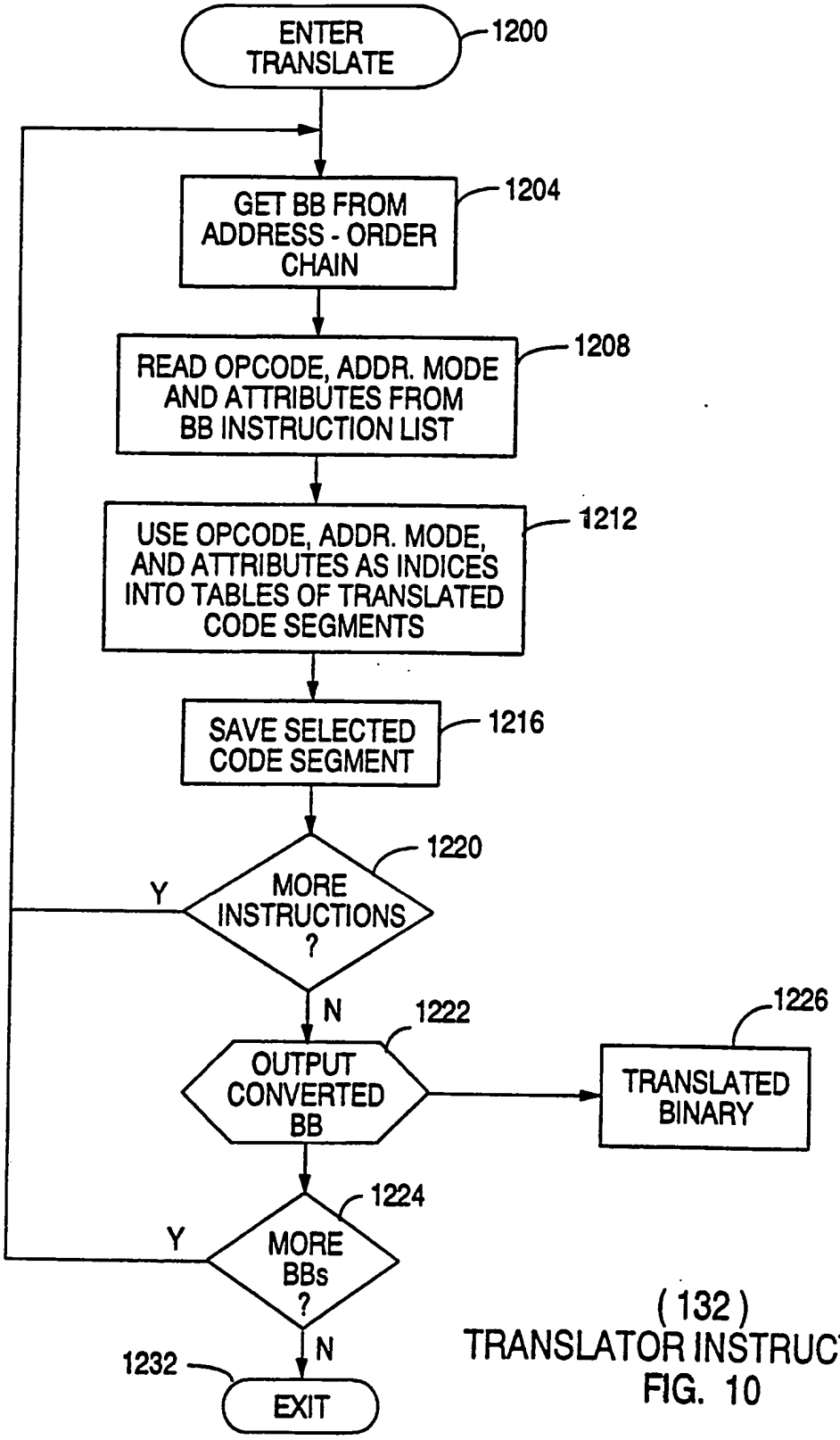
(316)
UNKNOWN ANALYSIS
FIG. 7



(124)
ANALYSIS FLOWGRAPH
FIG. 8



(812)
LIVE / DEAD ANALYSIS
FIG. 9



(132)
TRANSLATOR INSTRUCTIONS
FIG. 10

INTERNATIONAL SEARCH REPORT

International Application No. PCT/US89/02994

I. CLASSIFICATION OF SUBJECT MATTER (if several classification symbols apply, indicate all) [6]

According to International Patent Classification (IPC) or to both National Classification and IPC

IPC (4): G06F 5/00

U.S. Cl. 364/200

II. FIELDS SEARCHED

Minimum Documentation Searched [7]

Classification System	Classification Symbols
U.S.	364/200, 900

Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched [8]

III. DOCUMENTS CONSIDERED TO BE RELEVANT [9]

Category *	Citation of Document, [11] with indication, where appropriate, of the relevant passages [12]	Relevant to Claim No. [13]
A	U.S., A, 4,667,290 (GOSS) 19 MAY 1987, see entire document, especially Cols. 7 & 18	1,17,19, 20 & 21

* Special categories of cited documents: [10]

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

IV. CERTIFICATION

Date of the Actual Completion of the International Search	Date of Mailing of this International Search Report
18 Sept. 1989	11 OCT 1989
International Searching Authority	**Signature of Authorized Officer**
ISA/US	Lawrence E. Anderson

Form PCT/ISA/210 (second sheet) (Rev.11-87)